pess第十课作业

```
// figure 1
struct node_t
{
    data_t value;
    node_t *next;
};
struct queue_t
{
    node_t *head;
    node_t *tail;
    mutex_t h_lock;
    mutex_t t_lock;
};
void initialize(queue_t *q, data_t value)
{
    node_t *node = new_node(); // Allocate a new node
    node->value = value;
    node->next = NULL; // Make it the only node in the queue
    q->head = node;    // Both head and tail point to it
    q->tail = node;
    q->h_lock = FREE; // Locks are initially free
    q->t_lock = FREE;
}
void enqueue(queue_t *q, data_t value)
{
    node_t *node = new_node(); // Allocate a new node
    node->value = value;       // Copy enqueued value into node
    node->next = NULL;         // Set next pointer of node to NULL
    lock(&q->t_lock);          // Acquire t_lock to access tail
    q->tail->next = node;      // Append node at the end of queue
    q->tail = node;            // Swing tail to node
    unlock(&q->t_lock);        // Release t_lock
}
bool dequeue(queue_t *q, data_t *pvalue)
{
    lock(&q->h_lock);       // Acquire h_lock to access head
    node_t *node = q->head; // Read head
    new_head = node->next;  // Read next pointer
    if (new_head == NULL)
    {                       // Is queue empty?
```

```
41          unlock(&q->h_lock); // Release h_lock before return
42          return false;       // Queue was empty
43      }
44      *pvalue = new_head->value; // Queue not empty. Read value
45      q->head = new_head;        // Swing head to next node
46      unlock(&q->h_lock);        // Release h_lock
47      free_node(node);           // Free node
48      return true;               // Dequeue succeeded
49  }
50  // figure 2
51  struct pointer_t
52  {
53      node_t *ptr;
54      unsigned int count;
55  };
56  struct node_t
57  {
58      data_t value;
59      pointer_t next;
60  };
61  struct queue_t
62  {
63      pointer_t head;
64      pointer_t tail;
65  };
66  void initialize(queue_t *q, data_t value)
67  {
68      node_t *node = new_node();
69      node->value = value;
70      node->next.ptr = NULL;
71      q->head.ptr = node;
72      q->tail.ptr = node;
73  }
74  void enqueue(queue_t *q, data_t value)
75  {
76      node_t *node = new_node();
77      node->value = value;
78      node->next.ptr = NULL;
79      pointer_t tail;
80      while (true)
81      {
82          tail = q->tail;
83          pointer_t next = tail.ptr->next;
84          if (tail == q->tail)
85          {
86              if (next.ptr == NULL)// 比较tail.ptr->next 是否还为空，如果为空表示没有
87              {
```

```
88                  if (CAS(&tail.ptr->next, next, (struct pointer_t){node, next.cou
89                  {
90                      break;
91                  }
92              }
93              else // 否则表示有其他线程进行了插入操作
94              {
95                  CAS(&q->tail, tail, (struct pointer_t){next.ptr, tail.count + 1}
96              }
97          }
98      }
99      CAS(&q->tail, tail, (struct pointer_t){node, tail.count + 1});
100 }
101 // figure 3
102 bool dequeue(queue_t *q, data_t *pvalue)
103 {
104     pointer_t head;
105     while (true)
106     {
107         head = q->head;
108         pointer_t tail = q->tail;
109         pointer_t next = head.ptr->next;
110         if (head == q->head) // 判读目前位置head是否改变
111         {
112             if (head.ptr == tail.ptr) // 判读当前队列是否为空
113             {
114                 if (next.ptr == NULL) //为空说明队列已经空了，return false
115                 {
116                     return false;
117                 }
118                 CAS(&q->tail, tail, (struct pointer_t){next.ptr, tail.count + 1}
119             }
120             else
121             {
122                 *pvalue = next.ptr->value; //不知道作用
123                 if (CAS(&q->head, head, (struct pointer_t){next.ptr, head.count
124                 {
125                     break;
126                 }
127             }
128         }
129     }
130     free_node(head.ptr);//释放当前head
131     return true;
132 }
```

1. What are constraints on enqueue and dequeue in the FIFO queue? You do not need to look at the code yet

出队只能从队首出队，入队只能插入到队尾

2.What is the advantage of using two locks over one lock?

因为入队和出队操作互不影响，有两个锁的效率会更高。在代码中可以看到出队和入队操作，都只使用了自己的锁，不会相互影响。

3.In the style of comments of the lock-based FIFO queue code, add comments to the lock-free code (on paper), explaining what each line does. The comments should be short and precise (not more than 10 words each). We have provided you a copy of the code in Figure 2.

```
1  void initialize(queue_t *q, data_t value)
2  {
3      node_t *node = new_node(); //分配一个新的节点
4      node->value = value; // 给新节点赋值
5      node->next.ptr = NULL; // 设置新节点指向下一个节点为NULL
6      q->head.ptr = node; // 设置队头的地址
7      q->tail.ptr = node; // 设置队尾地址
8  }
9  void enqueue(queue_t *q, data_t value)
10 {
11     node_t *node = new_node(); // 分配一个新的节点
12     node->value = value; // 给新节点赋值
13     node->next.ptr = NULL; // 设置新节点指向下一个节点为NULL
14     pointer_t tail; //声明一个点
15     while (true)
16     {
17         tail = q->tail;//该点指向队尾
18         pointer_t next = tail.ptr->next;// 声明一个指向队尾下一个节点的点
19         if (tail == q->tail)//判断tail点与队列的尾节点是否相等
20         {
21             if (next.ptr == NULL)// 比较tail.ptr->next 是否还为空，如果为空表示没有
22             {
23                 if (CAS(&tail.ptr->next, next, (struct pointer_t){node, next.cou
24                 {
25                     break;//完成入队，跳出循环
26                 } //疑问：为啥这里不做下面的tail的比较
27             }
28             else // 否则表示有其他线程进行了插入操作
29             {
30                 CAS(&q->tail, tail, (struct pointer_t){next.ptr, tail.count + 1}
31             }
32         }
33     }
```

```
34      CAS(&q->tail, tail, (struct pointer_t){node, tail.count + 1});// 此时判断q->t
35  }
```

4. Explain how a new node is inserted into the lock-free queue. How many successful CASes are needed per node? What happens if the CAS in line 96 fails? How far can the tail lag behind? Is the program correct without line 96?

1.Explain how a new node is inserted into the lock-free queue. How many successful CASes are needed per node?

```
1  if (next.ptr == NULL)// 比较tail.ptr->next 是否还为空，如果为空表示没有其他线程进行修i
2   {
3      if (CAS(&tail.ptr->next, next, (struct pointer_t){node, next.count + 1})) //
4      {                                                                          //
5          break;//完成入队，跳出循环
6      }
7    }
8     else // 否则表示有其他线程进行了插入操作
9     {
10       CAS(&q->tail, tail, (struct pointer_t){next.ptr, tail.count + 1}); // 此时
11     }
```

至少一次，只要第3行的CAS执行成功就行了。

2.What happens if the CAS in line 96 fails?

如果96行返回的是failed，表示再其他线程的第92行进行了该值得插入，不会影响队列的结果。

3.How far can the tail lag behind?

在极端情况下，可能会一直增加

4.Is the program correct without line 96?

没有也能够正常运行。（只有在单线程，只插入一个值时，指向队尾的指针指向的是当前的队尾，没有指向新的队尾）

5. Carefully look at the code for the lock-free dequeue operation and answer the following questions:

(a) Line 104 checks what was already assigned in line 101. Why do we need line line 104?

这个检测表示当前的队列的头结点还是之前的头结点，没有被其他线程给出队

(b) In line 111 the value of the node is read before the head is updated in line 112. Why is this important? What can happen if we change the order of the lines?

*pvalue = next.ptr->value;这里不能够进行更换位置。不换位置前：CAS成功时，能够表示当前的pvalue的值还是之前的没有更改，保持了一致性。更换位置后：CAS成功了，在进行break之前进行赋

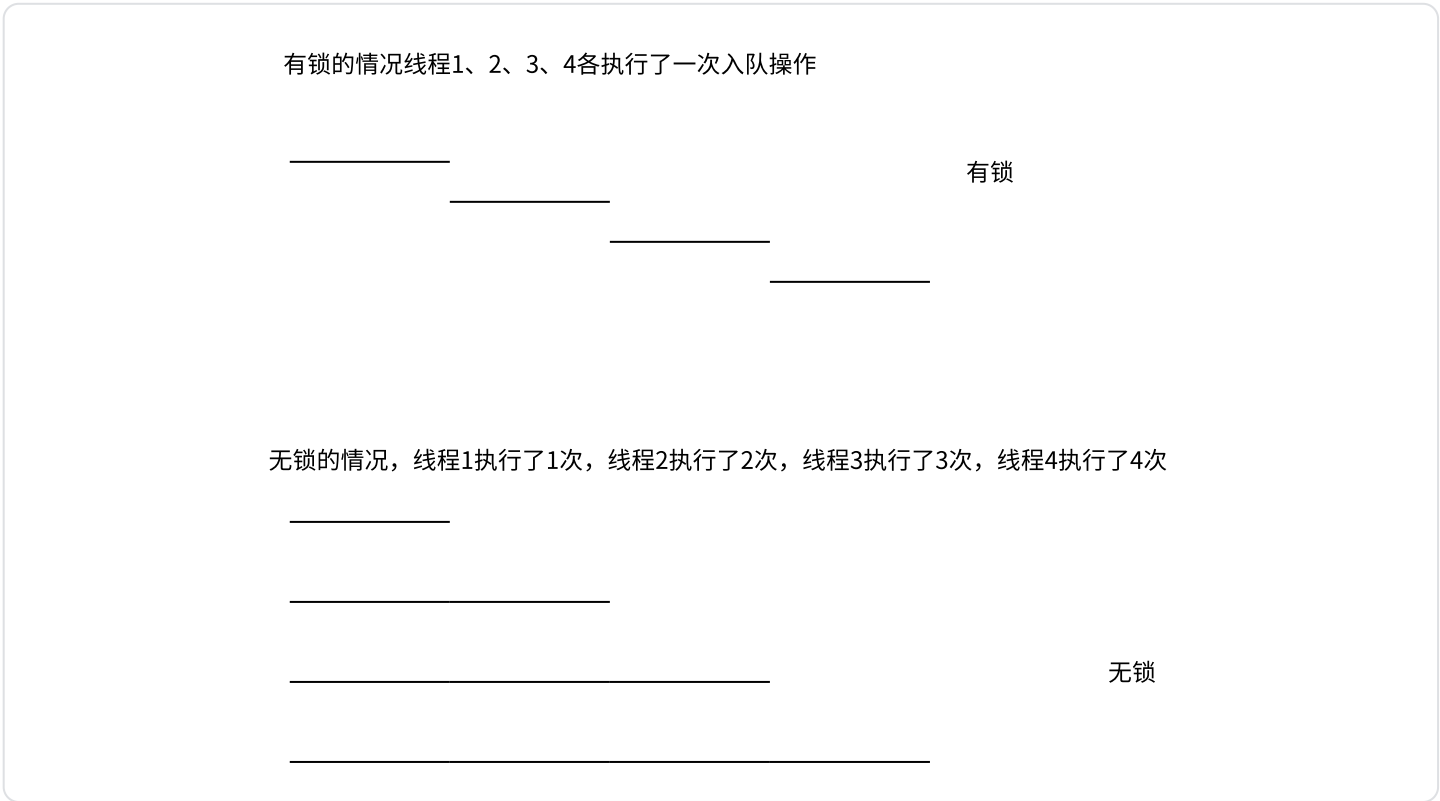
值的话，不能够保证当前的next.ptr->value的值，没有被其他线程给更改，不能够保持一致。

(c) What happens if the CAS in line 112 is unsuccessful?

如果112行没有成功则表示在其他线程进行了一次出队操作，那么该在循环一次，进行了出队操作

6. Which implementation do you expect to run faster — the lock-based or the lock-free? Explain your answer in terms of cost of the synchronization primitives, contention, synchronization overhead, etc.

我认为无锁的会更快。当多线程同时入队时，有锁的会抢占锁，其他线程会进入等待状态，等待锁的释放。而无锁的情况是当同时入队时，有一个线程提前入队成功，那么其他的线程会进入下一次的循环继续进行入队操作。直到所有线程跑完。因为是多线程并行，可能无锁执行次数更多，但最终时间是一致的。但是有锁的情况增加了加锁和解锁的操作耗时。



7. Show how to simplify the lock-based code if only one thread may enqueue nodes to the queue. Write the pseudocode and comment it. Explain in your own words why your solution is correct (i.e. any execution sequence keeps the FIFO ordering).

```
1 void enqueue(queue_t *q, data_t value)
2 {
3     node_t *node = new_node(); // Allocate a new node
4     node->value = value;       // Copy enqueued value into node
5     node->next = NULL;         // Set next pointer of node to NULL
6     q->tail->next = node;      // Append node at the end of queue
7     q->tail = node;            // Swing tail to node
8 }
```

8. Show how to simplify the lock-free code if only one thread may dequeue nodes from the queue. Write the pseudocode and comment it. Explain in your own words why your solution is correct (i.e. any execution sequence keeps the FIFO ordering) and why it is non-blocking.

```
bool dequeue(queue_t *q, data_t *pvalue)
{
    node_t *node = q->head; // Read head
    new_head = node->next;  // Read next pointer
    if (new_head == NULL)
    {                       // Is queue empty?
        return false;       // Queue was empty
    }
    *pvalue = new_head->value; // Queue not empty. Read value
    q->head = new_head;        // Swing head to next node
    free_node(node);           // Free node
    return true;               // Dequeue succeeded
}
```

9. Explain how count is used to handle the ABA problem discussed in recitation.

ABA问题大致是，在执行某种操作时，发生了故障，同时起了AB两个线程去做一件事，当A线程对value进行了+1操作，在B线程进行check前，刚好有一个C线程去将value值又进行了-1操作。当B线程进行check时，发现值没有问题，就又进行了一次+1操作，导致值与实际不符。

当我们在进行CAS比较时，我们新增了一个count，它能够在比较时，除了比较value外，还能够比较一下count值，当count不等时，说明有其他线程对这个做了操作了，不会再进行更改，避免了ABA问题。